BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images may include (but are not limited to):

- BLACK BORDERS
- TEXT CUT OFF AT TOP, BOTTOM OR SIDES
- FADED TEXT
- ILLEGIBLE TEXT
- SKEWED/SLANTED IMAGES
- COLORED PHOTOS
- BLACK OR VERY BLACK AND WHITE DARK PHOTOS
- GRAY SCALE DOCUMENTS

IMAGES ARE BEST AVAILABLE COPY.

As rescanning documents *will not* correct images, please do not report the images to the Image Problem Mailbox.

JAPIO

1/1 JAPIO - (C) JPO
PN - JP 11242597 A 19990907 [JP11242597]
TI - METHOD FOR GENERATION OF FLOW GRAPH OF JAVA BYTE CODE DATA
IN - GHOSH SANJOY
PA - SUN MICROSYST INC
AP - JP31391098 19980929 [1998JP-0313910]
PR - US97 940212 19970930 [1997US-0940212]
IC1 - G06F-009/45
AB - PROBLEM TO BE SOLVED: To optimize both velocity and efficiency of a Java optimization device and a JIT compiler by successively executing the forward and backward scanning of a byte code block and generating a link to a preceding or subsequent node in a file.
- SOLUTION: Forward scanning of a byte code source file 102 is executed in order to decide respectively the start codes 106, 108 and 110 of byte codes 112, 114 and 116 in a block that is not interrupted (100). Then, backward scanning of the file 102 is executed. In the code 116 is an instruction POP the POP is read and a node 120 is generated in a data flow graph including a POP instruction 122. Successively, a value 2 is read by the code 108 and linked to the instruction 122 (118).
-

Search statement 20

?

(19)日本国特許庁（ＪＰ）　(12)公開特許公報（Ａ）　(11)特許出願公開番号

特開平11－242597

(43)公開日　平成11年(1999)９月７日

(51)Int.Cl.[6]	識別記号	ＦＩ	
Ｇ０６Ｆ　9/45		Ｇ０６Ｆ　9/44	３２２Ｆ

審査請求　未請求　請求項の数２　ＦＤ　外国語出願　（全 33 頁）

(21)出願番号　特願平10－313910

(22)出願日　平成10年(1998)９月29日

(31)優先権主張番号　０８／９４０２１２

(32)優先日　1997年９月30日

(33)優先権主張国　米国（ＵＳ）

(71)出願人　595034134
サン・マイクロシステムズ・インコーポレイテッド
Ｓｕｎ　Ｍｉｃｒｏｓｙｓｔｅｍｓ，　Ｉｎｃ．
アメリカ合衆国　カリフォルニア州　94303　パロ　アルト　サン　アントニオ　ロード　901

(72)発明者　サンジョイ・ゴシュ
アメリカ合衆国94086カリフォルニア州サニーベイル、バイセント・ドライブ1235番、ナンバー54

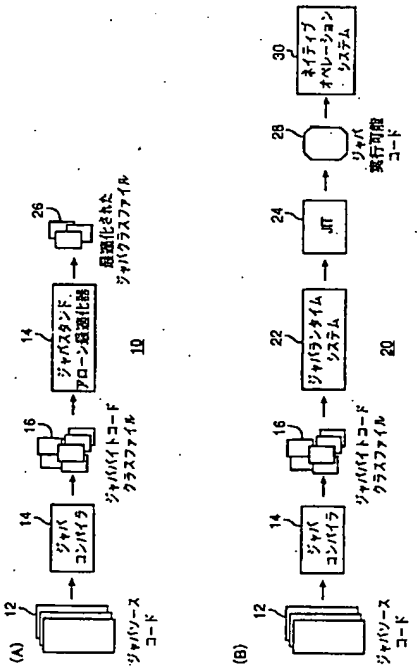
(74)代理人　弁理士　青山　葆　（外１名）

(54)【発明の名称】　ＪＡＶＡバイトコードデータのフローグラフの生成方法

(57)【要約】　（修正有）

【課題】ジャバ最適化器とJITコンパイラの速度と効率を最適化するIRを形成する為ソースバイトコードからDFGを生成する方法を提供する。

【解決手段】バイトソースファイルからの中断されないバイトコードブロックの前方向スキャンを中断されないブロック内各バイトコード開始を同定する為に実行し、次に後方向スキャンを実行する。ブロック内各バイトコードについてDFGノードを生成し、ブロック内バイトコードによって使用される全ての値に対するリンクを達成し、ファイル内の先行する又は後続するノードに対するリンクを生成する。



1

【特許請求の範囲】

【請求項１】　データフローグラフの形成において、中断されることのないバイトコードブロックのバイトコードをリンクさせる方法は以下のステップからなる：上記バイトコードの各々の開始を同定するため、中断されることのないバイトコードブロックを前進方向にスキャンする；上記中断されることのないバイトコードブロック内において、上記バイトコードの各々をバイトコード単位で後進方向にスキャンする；および上記バイトコードの各々によって使用される上記バイトコードの他の全てに対して、上記バイトコードの各々をリンクさせる、上記データフローグラフ内のリンクを生成する。

【請求項２】　データフローグラフの形成において、中断されることのないバイトコードブロック間でバイトコードをリンクさせる方法であって、上記中断されることのないバイトコードブロックは、上記中断されることのないブロックの実行順にしたがったリンクを有し、スタック状態は上記中断されることのないバイトコードブロックの各々について生成される方法は、以下のステップからなる：上記中断されることのないブロックの一つにおいて、ある値を生成する各バイトコードと上記中断されることのないブロックの第１経路上のいま一つにおいて上記値を消費する各バイトコード間に上記データフローグラフにおけるリンクを生成するため、上記実行順の複数の経路のうち、ジョインで終端する第１の経路を通して歩進する；およびある値を生成する各バイトコードのための上記第１の経路以外の他の全ての経路において、上記第１の経路内における上記中断されることのないブロックの１つにある値を生成する各バイトコードと同様のスタック位置を有する各バイトコードのためのリンクを用いて、上記第１の経路内の各リンクを複写する。

【発明の詳細な説明】

【０００１】

【発明の属する技術分野】本発明はジャバ（Ｊａｖａ）クラスファイルの最適化に関する。より詳細には、本発明はバイトコードのブロック内およびバイトコードのブロック間におけるデータ値の伝搬を同定するジャバクラスファイル最適化器からのデータフローグラフのアスペクト（ａｓｐｅｃｔ）の生成に関する。

【０００２】

【従来の技術】ソフトウエア開発者およびコンピュータのユーザにとって知られている１つの問題は、オペレーションシステムのプラットフォーム間におけるソフトウエアの可搬性の欠如である。この問題に向けた試みは、ワールドワイドウェブの如き膨張するネットワークにおいてコンピュータが相互にリンクされるにしたがって安全性を保証する手段を含む必要がある。これらの両方に対して対応に関して、複数層の安全性保護を含むべくデザインされたプラットフォームに独立なオブジェクト指

2

向のコンピュータ言語として、ＪＡＶＡプログラム言語がサンマイクロシステムズによって開発された。

【０００３】ジャバは、コンパイル言語と翻訳言語の両方であることによってオペレーションシステムに対する独立性を達成している。最初に、ジャバクラスファイルからなるジャバソースコードがジャババイトコードと呼ばれる汎用中間フォーマットにコンパイルされる。ジャババイトコード単一バイトオブコード（ｏｐｃｏｄｅ）のシーケンスによって構成され、各オブコードは実行されるべき特定のオペレーションを同定する。更には、オブコードの幾つかはパラメータを有する。オブコード番号２１　ｉｌｏａｄ＜ｖａｒｎｕｍ＞は局所変数ｖａｒｎｕｍ内に格納された単一ワードの整数値を取り出してスタック上にプッシュする。

【０００４】次に、バイトコードは、ジャババーチャルマシーン（Ｊａｖａ　Ｖｉｒｔｕａｌ　Ｍａｃｈｉｎｅ　ＪＶＭ）によって解釈され、バイトコードが固有の（ｎａｔｉｖｅ）マシーンコードに翻訳される。ＪＶＭは物理的なマイクロプロセッサと多くの特徴を共有する“仮想”のプロセッサのスタック化ベースの具体化である。ＪＶＭによって実行されるバイトコードは本質的にはマシーンインストラクションの集合であり、当業者によって認識されるであろうが、計算機のアッセンブリイ言語に類似している。したがって、各ハードウエアプラットフォームもしくはオペレーションシステムは、ジャバランタイムシステム（Ｊａｖａ　Ｒｕｎｔｉｍｅ　Ｓｙｓｔｅｍ）と呼ばれる、ＪＶＭの固有の具体化手段を有することができ、ユニバーサルなバイトコード呼出しを下層の固有システムにルート付けする。

【０００５】

【発明が解決しようとする課題】ＪＡＶＡバイトコードのインストラクションセットを開発するに当たって、設計者は、ハードウエアの最適化にとって十分に簡単であることとともに、安全性保護を与え、不適切にフォーマットされたバイトコードに帰因する実行エラーやシステム破壊を防ぐ検証手段が含まれているということを保証することを要求された。ジャババイトコードは有意型（ｓｉｇｎｉｆｉｃａｎｔ　ｔｙｐｅ）の情報を含むので、検証手段は、バイトコードがインターネットや局所のディスクから最初に取り出されたときに、広範囲型（ｅｘｔｅｎｓｉｖｅ　ｔｙｐｅ）のチェックを実行することができる。その結果、ネイティブマシーンのインタープリータは、実行時に最小限の（ｍｉｎｉｍａｌ　ｔｙｐｅ）チェックを実行するのみでよい。広範囲な演算時チェックを実行することによって保護を与えるＳｍａｌｌ　Ｔａｌｋの如き言語とは異なり、ジャバは演算時チェックをより高速に実行することができる。

【０００６】ジャバは検証手段によって安全性を保証し、バイトコードによって可搬性（ｐｏｒｔａｂｉｌｉｔｙ）を与えるが、ジャバプログラムはＣ／Ｃ＋＋の如

き言語で書かれたネイティブにコンパイルされたプログラムをその実行時に遅延させる。ユーザがウエブページ（Ｗｅｂ　Ｐａｇｅ）上でジャバプログラムを立上げた場合、ユーザはプログラムがダウンロードされるのを待つのみでなく、翻訳されるまで待たなければならない。ジャバの実行時間を改善するためには、ジャババイトコードの処理に最適化を導入することができる。この最適化は、スタンドアローン最適化器（Ｓｔａｎｄ－Ａｌｏｎｅ　Ｏｐｔｉｍｉｚｅｒ　ＳＡＯ）として、或いはジャストインタイム（Ｊｕｓｔ－ｉｎ－Ｔｉｍｅ　ＪＩＴ）コンパイラの一部として含む様々の態様で具体化することができる。

【０００７】ＳＡＯはバイトコードを含む入力クラスファイルを、同じ演算をより効率的に実行するバイトコードを含む出力クラスファイルに変換する。ＪＩＴはバイトコードを含む入力クラスファイルを実行可能なプログラムに変換する。ＪＩＴの開発以前では、ＪＶＭがプログラム内の全てのバイトコードインストラクションをステップ処理して機械的にネイティブコードコールを実行していた。しかし、ＪＩＴコンパイラを用いる場合、ＪＶＭはまずＪＩＴに対し呼び出しをかけ、ＪＩＴはインストラクションをネイティブなオペレーションシステム上で直接に実行されるネイティブコードにコンパイルする。ＪＩＴコンパイラはネイティブにコンパイルされたコードがより高速に実行されることを可能にし、コードが一度だけコンパイルされればそれで済むことを可能にする。更に、ＪＩＴコンパイラは、実行可能なコードが最適化されうるステージを提供する。

【０００８】バイトコードの最適化、或いはコンパイルは、当業者には中間表現（ｉｎｔｅｒｍｅｄｉａｔｅ　ｒｅｐｒｅｓｅｎｔａｔｉｏｎ　ＩＲ）として知られているソースバイトコードへの翻訳を必要とする。ＩＲは、プログラムの２つの基本要素、即ち、制御フローグラフ（ＣＦＧ）およびデータフローグラフ（ＤＦＧ）に関する情報を与える。次いで、ＩＲはコンパイラに関してはオブジェクトコードに、最適化器の場合にはソースコードフォーマットの改善バージョンに変換される。

【０００９】ＣＦＧはコードを中断されることのないグループとして常に実行されるバイトコードブロックに分断し、ブロックを相互にリンクする関係を作り上げる。ＤＦＧは、値が生成される所とそれらが使用される所の間の関係をマップする。これはバイトコードブロック内の関係とバイトコードブロック間の関係とを含む。ジャバプログラムは、暗黙の変数としてスタック位置を多用するので、値の伝播のトラッキングは困難なプロセスである。ＩＲのためにＤＦＧを生成するため、データフローの情報を抽出する場合、従来のアプローチは全ての可能なフローパスを通じて、スタックと反復の両方の完全なシミュレーションを実行することであった。

【００１０】エンドユーザがコンパイルされたプログラムと協働するのみである伝統的なコンパイラとは対照的に、ジャバのランタイム特性は、ジャバの最適化器とＪＩＴコンパイラの速度と効率を最適化するための重要なインセンティブを創り出す。したがって、本発明の目的は、ジャバの最適化器とＪＩＴコンパイラの速度と効率を最適化するＩＲを形成するためソースバイトコードからＤＦＧを生成する方法を提供することである。本発明のこれらおよび他の多くの目的および利点は、発明の図面の考察および以下の記述から当業者にとって明かになるであろう。

【００１１】

【課題を解決するための手段】本発明は、ＩＲのためのＤＦＧの生成の効率を改善することによってジャバクラスファイルの最適化に向けられている。本発明の第１の態様によれば、バイトソースファイルからの中断されないバイトコードブロックの前方向スキャンは、中断されないブロック内の各バイトコードの開始を同定するために実行される。次に、後方向スキャンが実行される。ブロック内の各バイトコードについて、ＤＦＧノードが生成され、ブロック内のバイトコードによって使用される全ての値に対するリンクが達成され、ファイル内の先行する或いは後続するノードに対するリンクが生成される。

【００１２】本発明の第２の態様によれば、バイトコードブロックはファイルの実行順にリンクされるとともに、スタック状態が各コードブロックについて生成される。ＣＦＧによって生成される各加入ステートメントについて、ＣＦＧの１つのパスが歩進される。次に、ＤＦＧ内において、値が使用される個所と値が生成される個所との間にリンクが達成される。最後に、値を生成するブロックに平行な全てのブロックについて、同じスタック状態個所を専有するバイトコードのためのリンクが複写される。

【００１３】

【発明の実施の形態】当業者であれば、本発明の以下の記述は図示のためだけであり、いかなる意味においても制限的なものでないことを認識するであろう。本発明の他の実施例は、開示の範囲内における考察から当業者にとって容易に示唆されるであろう。

【００１４】まず、図１の（Ａ）及び（Ｂ）を参照すると、ジャババイトコードソースファイルの最適化の簡略化フローダイアグラム１０および２０が図示されている。両方のフローダイアグラム１０と２０においても、ジャバソースコードファイル１２はジャバコンパイラ１４によってジャババイトコードクラスファイル１６にコンパイルされる。フローダイアグラム１０では、クラスファイルはＳＡＯ１８において演算され、フローダイアグラム２０では、クラスファイルはジャバランタイムシステム２２を通してＪＩＴコンパイラ２４に渡される。ジャバＳＡＯ１８は、入力クラスファイルと同じ演算を

実行するバイトコードを含むジャバクラスファイル２６を最適化された方法においてのみ出力する。ＪＩＴ２４はネイティブのオペレーションシステム３０上で直接に実行される実行可能なコード２８を出力する。当業者ならば、図１に記述された手順は単に例示的であり、クラスファイルが最適化されうる他の構造が存在することを認識できるであろう。

【００１５】入力バイトコードソースクラスファイルの処理において、ＳＡＯとＪＩＴの両方がＩＲを形成する。上記した如く、ＩＲはＣＦＧとＤＦＧへのプログラムの制御およびデータフロー情報の簡単で直接的な構造化である。ＣＦＧはプログラムの実行のためにリンクされたコードブロックとしてバイトコードインストラクションの実行のシーケンスと順序に関する情報を表す。コードブロックは、中断されないグループとして常に実行されるバイトコードの区切りであり、コードブロック間のリンクはバイトコードブランチ、条件又はジャンプインストラクションである。

【００１６】図２はバイトコードのクラスファイル４０からのＣＦＧの生成を示すフローダイアグラムである。ＣＦＧの生成方法は複数知られていることが認識されるべきである。開示の重複的な複雑化およびそれにより本発明の不明確化を避けるため、これら方法のいずれかの詳細を開示することはしない。プロシージャは１からＮまで順番が付けられた順序付けバイトコードインストラクションシーケンスを含むクラスファイル４０を用いて開始される。１からＮの順番は、当業界において知られているように、コンピュータプログラムの行の番号付けとの類推である。次に、最適化器４２はバイトコードから制御フロー情報を抽出するためバイトコードクラスファイル４０を処理する。

【００１７】制御フロー情報は、ＣＦＧ４４によって図示されている。例えば、ＣＦＧ４４は、クラスファイル４０について、バイトコードインストラクション１から３を最初に実行し、インストラクション１００から１０３を次に実行し、インストラクション１０から１４又は１５から１９のいずれかを実行し、最後にインストラクション４から８が残りのバイトコードインストラクションの実行の前に実行されるべきことを指示する。ＣＦＧ４４は単なる例であり、最適化器４２によってクラスファイル４０からいかなる数のＣＦＧを作ってもよいことが理解されるであろう。

【００１８】上記の如く、ＣＦＧが一旦生成されると、ＤＦＧが生成される。本発明によれば、ＤＦＧの２つの態様がＣＦＧから生成される。１つの態様ではなく、バイトコードのブロック内における値の伝播が決定される。典型的なＪＡＶＡプログラムにおいて知られているように、データ値は１個所において生成され、他の個所で使用される。本発明の第１の態様は、中断されることのないコードブロック内において発生する生成点と消費点の間の関係を樹立することに関係する。本発明は暗黙の変数が容易に認識され、それらの値がＤＦＧ内においてそれらを用いるバイトコードに直ちにリンクされることを可能にする。

【００１９】先行技術とは異なり、本発明はバイトコードによって使用される値を決定するためスタックを構成する先行技術において採用されたアプローチを避ける。例えば、図３にはＣＦＧのコードブロック内においてリンクを生成するための公知の方法のブロックダイアグラムが示されている。このダイアグラムはＤＦＧを生成するに際して、暗黙のスタック位置変数の値を決定するため採用された主要なステージを例示している。ステージ６０の間、バイトコード６２の入力シーケンスの演算がシーケンスが表すスタック６４のあるバージョンを生成するためにシミュレートされる。６４で示されるスタックはバイトコード６２の入力シーケンスの一部分６６のみを示す。ステージ６８の間、スタック６４が実行され、ＰＯＰインストラクション７０が先行するＢＩＰＵＳＨ“２”インストラクション７２によって値“２”と関係付けられることが決定される。この情報は参照番号７４と７６によって表されるＤＦＧノード内に格納される。

【００２０】本発明の好ましい実施例によれば、ノードツリーが採用される。バイトコードクラスファイルによって値が生成されると、新しいノードがツリー内に生成される。このノードは公知になるように、これを値の生成および消費点にリンクさせることによってツリーの他の部分に関係付けられる。この方式において、プログラムを通してのデータ値の伝播は容易に追跡することができる。当業者ならば、種々のデータ構造を使用しうることを認識するであろう。

【００２１】図４に戻って、本発明による、バイトコードブロック内におけるリンクの生成方法のフローダイアグラムが図示されている。第１ステップ１００では、バイトコードソースファイル１０２の前進方向スキャンは、公知の如くバイトコードは異なる長さでありうるので、バイトコード１１２、１１４および１１６の開始（スタート）１０６，１０８および１１０を夫々決定するために実行される。

【００２２】次に、ステップ１１８において、バイトコードソースファイル１０２の後進方向スキャンが実行される。後方向スキャンステップ１１８では、ＤＦＧのノードがそれらが発生するにしたがって生成される。例えば、バイトコード１１６がインストラクションＰＯＰであれば、ＰＯＰが読取られるとノード１２０がＰＯＰインストラクション１２２を含むＤＦＧ内に生成される。次に、インストラクションＢＩＰＵＳＨ“２”がバイトコード１０８で読取られると、１２４において、値“２”がそれを使用するＰＯＰインストラクション１２２にリンクされる。

【００２３】本発明は、スタックのコピーの構築全体のプロセスがなくなるので、ＤＦＧの生成における効率の大幅な増加を与える。本発明は、バイトコードのストリームはスタックマシーンに対するインストラクションであるので、バイトコードインストラクションによって消費されるオペランドあるいは値がインストラクションに先行することを認識している。バイトコードファイルが逆順に読まれるときに、ＤＦＧ内のノードを生成するステップはバイトコードが読まれた後、直ちに実行される。

【００２４】全体をカバーする意図ではないが、更なる例が本発明によるバイトコードインストラクションから生成されるＤＦＧ内のノードについて、図５（Ａ）と（Ｂ）に図示されている。図５（Ａ）と（Ｂ）は、夫々２つのリンクを持った単一インストラクションおよび多数のリンクを持ったインストラクションのシリーズの例である。

【００２５】図５（Ａ）において、バイトコードソースブロック１５０は、まず、前進方向スキャンによってバイトコード１５２、１５４および１５８に分けられる。後進方向のスキャンステップが実行される時に、整数加算（ＩＡＤＤ）インストラクションがバイトコード１５８から読取られるとともに、ノード１６０がＤＦＧ内に生成される。インストラクションＩＡＤＤの定義からノード１６０は２つの値を消費することが知られている。後進方向スキャンが継続すると、これら２つの値がバイトコード１５４と１５２から読取られ、ＤＦＧのノードツリー内のノード１６２と１６４として生成され、ノード１６０にリンクされる。

【００２６】図５（Ｂ）において、バイトコードソースブロック１６６は前進方向スキャンによってバイトコード１６８、１７０および１７２にまず区分される。バイトコード１６６のシーケンスはステートメント（Ｘ＋Ｘ）を実施する。後進方向スキャンステップでは、最初に出会ったバイトコード１７２はＩＡＤＤインストラクションである。図５（Ａ）の例に関して説明したように、ＩＡＤＤインストラクションはＤＦＧ内において２つの値とのリンクを持ったノード１７４を生成する。次のバイトコード１７０は複写（ＤＵＰ）インストラクションであるので、ノード１７４からの両方のリンクはＤＵＰインストラクションのためノード１７６に対して作られる。次に、整数ロードＸ（ＩＬＯＡＤＸ）インストラクションを有するバイトコード１６８が読まれ、ノード１７６に対するリンクを持ったノード１７８が生成される。ノードツリーにおけるこのシーケンスにおいて、ノード１７８の値はノード１７６に複写され、両方の値はノード１７４において消費される。上に掲げた例から当業者には明かなように、後進方向スキャンステップを実行する際、生成された特定のＤＦＧはＣＦＧによって生成されたブロック内のバイトコードのシーケンスに依存する。

【００２７】本発明の第２の態様によれば、ＣＦＧ内のバイトコードブロック間の値の伝播が決定される。図６はリンクされたコード２００、２０２、２０４および２０６の複数のブロックを持ったＣＦＧ１８０を示している。ＣＦＧ１８０ではジョイン（ｊｏｉｎ）２０８に先行するブロック２０２と２０６は平行であると考えられる。ジョインは複数の平行なブロックを接続することであると考えられるべきである。ＣＦＧが生成されると、スタック状態２１０、２１２、２１４および２１６がバイトコード２００、２０２、２０４および２０６の各ブロックについて夫々生成される。スタック状態は当該スタックに関係するブロック内のバイトコードが実行された後に、スタック上に依然存在する全ての値を含む。スタック上の値はバイトコードブロック内のインストラクションによって生成されるということを認識すべきである。このことは、インストラクション２１８と２２０のスタック状態２２２と２２４への夫々のマッピングによって図示されている。以下の記述の図示の目的で、インストラクション２１８はＰＵＳＨ“１”であり、インストラクション２２０はＰＵＳＨ“０”であり、インストラクション２２６はＰＯＰである。

【００２８】バイトコードの始めのブロック内に生成された値がバイトコードのブロック間においてＤＦＧ内のリンクを生成するためにバイトコードの相続くブロックによって使用される又は消費される時点を決定するため、先のスタック状態値の消費を注目するとともに、そのソースを同定しながら、ブロックの実行がなされる。この情報はバイトコードの消費と生成のノード間のリンクとしてＤＦＧ内に表示される。全ての平行なブロックのためのリンクを樹立するため、従来の方法はＣＦＧ内における全ての可能な制御フロー経路を通しての繰返し（ｉｔｅｒａｔｉｎｇ）に依存していた。例えば、ＣＦＧ１８０内のブロック間のリンクについてＤＦＧを生成するため、第１経路はブロック２００、２０２および２０４を含む経路を通して作られ、次いで、第２経路がブロック２００、２０６および２０４を含む経路を通して作られる。

【００２９】本発明によれば、バイトコードの平行ブロック間のリンクについてＤＦＧを形成するために全ての制御フロー経路を検証する必要はない。ブロック２００、２０２、２０４および２０６間のＤＦＧ内のリンクは、ブロック２００、２０２、２０４を通して歩進し、その後、ブロック２０６に対するリンクを樹立するために、ブロック２００，２０２、２０４を通した経路からの情報を適用することによって生成される。本発明は、ジャバの標準要求および上記した検証は適切にフォーマットされたコードではジョインに先行するブロックのいずれのスタック状態も同一であることを保障するという事実を活用している。

【００３０】例えば、ＣＦＧ１８０において、ジョイン２０８に先行するスタック状態２１２と２１６の両方はサイズおよび相対形式の両方において同一でなければならない。実行時、ブロック２０２内のある値の消費はブロック２０６内のある値の消費と平行に行なわれる。その結果、例えば、スタック状態位置２２２に対応する値がコードブロック２０４の２２６におけるバイトコードインストラクションによって使用されるならば、バイトコードインストラクション２２６はスタック状態位置２２４に対応する値をも使用する。

【００３１】シュードコード（ｐｓｅｕｄｏｃｏｄｅ）の以下の行は点を示している。もし（ＩＦ）（何らかの条件が発生したら）

Ｘ＝１；

その他（ＥＬＳＥ）

Ｘ＝０；

：

：

Ａ＝Ｘ＋２

【００３２】式“Ａ＝Ｘ＋２”における“Ａ”に対する値の付与には、“ＩＦ（何らかの条件が発生すれば）”ステートメントによって決められるような暗黙のスタック変数“Ｘ”として割当てられた値を“Ｘ＝１”又は“Ｘ＝０”のいずれかによって用いる。

【００３３】図７（Ａ）において、本発明の第２の態様にしたがってＤＦＧを生成するために採用されたステップを概観するブロックダイアグラムが図示されている。ステップ２５０において、ＣＦＧグラフの第１のブランチがスキャンされる。図６のＣＦＧ１８０に適用すると、ブロック２００、２０２および２０４を通る第１経路が実行される。

【００３４】次に、ステップ２５２において、値が生成される個所と値が消費される個所との間のリンクが形成される。図７（Ｂ）に図示されているように、リンク２５４がＣＦＧ１８０内の位置２１８におけるＰＵＳＨ“１”インストラクションとＣＦＧ１８０内の位置２２６においてこれを消費するＰＯＰインストラクションとの間に形成される。

【００３５】次に、ステップ２５６において、ＰＵＳＨ０インストラクションのためのデータはＣＦＧ１８０内の位置２１８のＰＵＳＨ１インストラクションと同じスタック状態位置を占有することが知られているので、リンク２５８が位置２２０におけるＰＵＳＨ０インストラクションとこれを消費する位置２２６におけるＰＯＰインストラクションとの間に形成される。このようにして、本発明の第２の態様によれば、従来では行なわれていたＣＦＧ内の全ての経路を検証することを要することなく、バイトコードブロック間のＤＦＧにおけるリンクはより効率的に形成される。ジョインはバイトコードソースコード全体を通じて極めて共通であるので、本発明は効率における大幅な増加を実現する。多重ジョインが存在する場合には、効率の増加はより大きい。

【００３６】この発明の例示的な実施例と応用が図示され、記述されたが、上に述べたものよりもより多くの修正がここに設定した発明思想から逸脱することなしに可能であることが当業者にとって明らかであろう。本発明は、それ故、添付の特許請求の範囲の技術的思想の範囲内における外、制限されるべきではない。

【図面の簡単な説明】

【図１】　（Ａ）は、本発明によるスタンドアローン最適化器を用いたＪＡＶＡバイトコードソースファイルの最適化のフローダイアグラムであり、（Ｂ）は本発明によるジャストインコンパイラを用いたＪＡＶＡバイトコードソースファイルの最適化のフローダイアグラムである。

【図２】　本発明において使用するのに好適なバイトコードソースファイルからの制御フローグラフの生成を図示するフローダイアグラムである。

【図３】　先行技術によるデータフローグラフの生成のフローダイアグラムである。

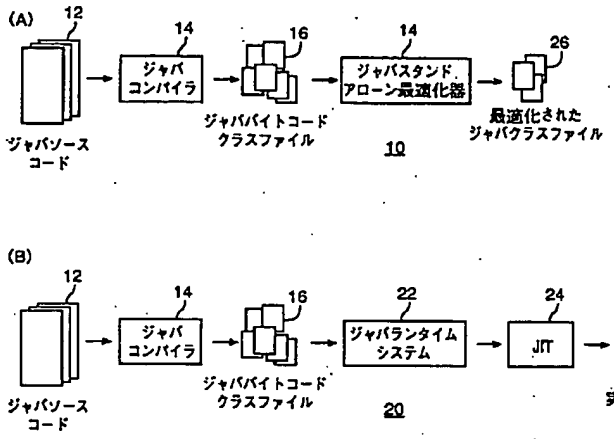
【図４】　本発明によるバイトコードブロック内のリンクのためのデータフローグラフの生成の第１の態様のフローダイアグラムである。

【図５】　（Ａ）は本発明の第１の態様によるインストラクションＩＡＤＤのためのデータフローグラフ内のノードの生成についてのブロックダイアグラムであり、（Ｂ）は本発明の第１の態様による複数のインストラクションについてデータフローグラフ内の複数のノードの生成についてのブロックダイアグラムである。

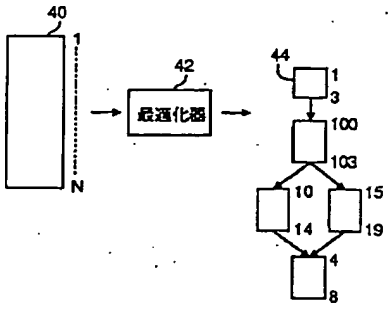
【図６】　本発明によるバイトコードブロック間のリンクのためのデータフローグラフの生成の第２の態様を図示するのに好適な制御フローグラフのフローダイアグラムである。

【図７】　（Ａ）は本発明の第２の態様にしたがった方法におけるステップのブロックダイアグラムであり、（Ｂ）は本発明の第２の態様によるデータフローグラフにおけるノードの生成のためのブロックダイアグラムである。

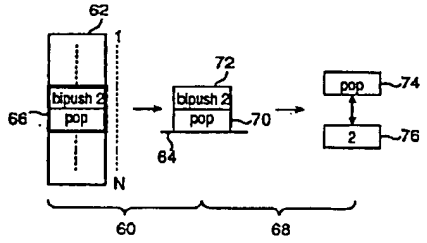
【図1】



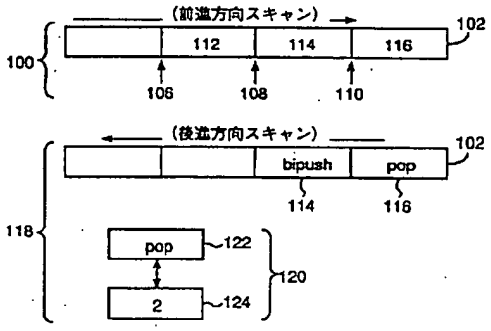
【図2】



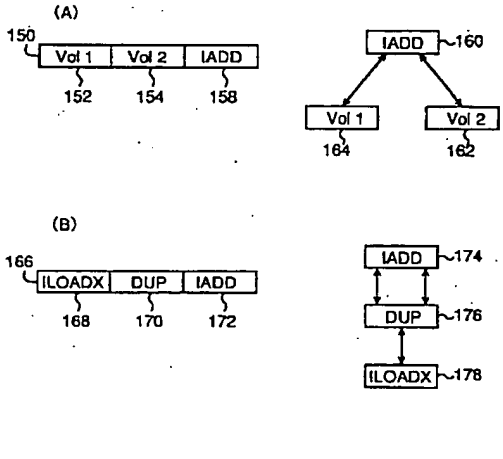
【図3】



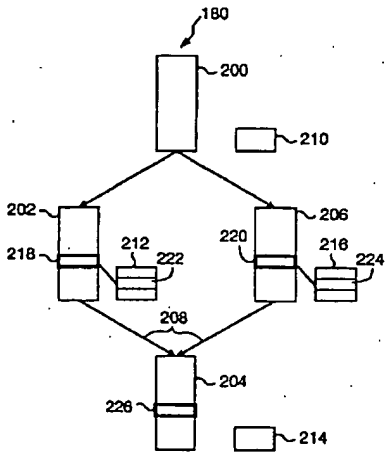
【図4】



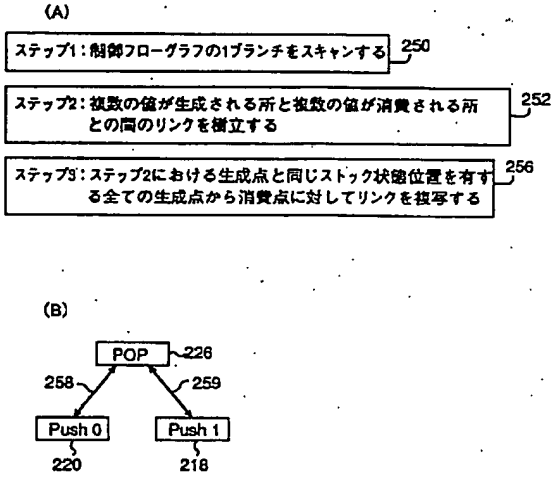
【図5】



【図6】



【図７】


(A)
ステップ1：制御フローグラフの1ブランチをスキャンする
250
ステップ2：複数の値が生成される所と複数の値が消費される所との間のリンクを樹立する
252
ステップ3：ステップ2における生成点と同じストック状態位置を有する全ての生成点から消費点に対してリンクを複写する
256
(B)
POP
226
258
259
Push 0
Push 1
220
218

【外国語明細書】

SUNP-2412

This application is submitted in the name of inventor Sanjoy Ghosh, assignor to Sun Microsystems, Inc., a California Corporation.

SPECIFICATION

METHOD FOR GENERATING A JAVA BYTECODE DATA FLOW GRAPH

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention relates to the optimization of Java class files. More particularly, the present invention relates to the generation of aspects of a data flow graph from a Java classfile optimizer that identifies the propagation of data values in blocks of bytecode and between blocks of bytecode.

2. The Prior Art

A known problem for software developers and computer users is the lack of portability of software across operating system platforms. Attempts to address this problem must include means of ensuring security as computers are linked together in ever expanding networks, such as the World Wide Web. As a response to both of these concerns, the JAVA programming language was developed at Sun Microsystems as a platform independent, object oriented computer language designed to include several layers of security protection.

整理番号 １６２８８０

SUNP-2412

Java achieves its operating system independence by being both a compiled and interpreted language. First, Java source code, which consists of Java classfiles, is compiled into a generic intermediate format called Java bytecode. Java's bytecodes consist of a sequence of single byte opcodes, each of which identify a particular operation to be carried out. Additionally, some of the opcodes have parameters. For example, opcode number 21, *iload <varnum>*, takes the single-word integer value stored in the local variable, *varnum*, and pushes it onto a stack.

Next, the bytecodes are interpreted by a Java Virtual Machine (JVM) which translates the bytecodes into native machine code. The JVM is a stacked-based implementation of a "virtual" processor that shares many characteristics with physical microprocessors. The bytecodes executed by the JVM are essentially a machine instruction set, and as will be appreciated by those of ordinary skill in the art, are similar to the assembly language of a computing machine. Accordingly, every hardware platform or operating system may have a unique implementation of the JVM, called a Java Runtime System, to route the universal bytecode calls to the underlying native system.

When developing the JAVA bytecode instruction set, the designers sought to ensure that it was simple enough for hardware optimization and also included verification means to provide security protection and to prevent the execution errors or system crashes that can result from improperly formatted bytecode. As Java's bytecodes contain significant type information, the verification means are able to do extensive type

checking when the bytecodes are first retrieved from the internet or a local disk. As a result, the interpreter of the native machine need only perform minimal type checking at run time. Unlike languages such as SmallTalk that provide protection by performing extensive runtime checks, Java executes more quickly at run time.

Although Java provides security through verification means and portability through bytecodes, Java programs lag natively compiled programs, written in languages like C/C++, in their execution time. When a user activates a Java program on a Web Page, the user must wait not only for the program to download but also to be interpreted. To improve Java's execution time, optimizations can be introduced into the processing of Java bytecodes. These optimizations can be implemented in a variety of manners including as Stand-Alone Optimizers (SAO) or as part of Just-in-Time (JIT) compilers.

A SAO transforms an input classfile containing bytecode into an output classfile containing bytecodes that more efficiently perform the same operations. A JIT transforms an input classfile containing bytecode into an executable program. Prior to the development of JITs, a JVM would step through all the bytecode instructions in a program and mechanically perform the native code calls. With a JIT compiler, however, the JVM first makes a call to the JIT which compiles the instructions into native code that is then run directly on the native operating system. The JIT compiler permits natively complied code to run faster and makes it so that the code only needs to be compiled once. Further, JIT compilers offer a stage at which the executable code can be

optimized.

To either optimize or compile bytecodes involves the translation of the source bytecodes into what is known in the art as an intermediate representation (IR). The IR provides information about two essential components of a program: the control flow graph (CFG) and the data flow graph (DFG). Subsequently, the IR is transformed for compilers into object code and for optimizers into an improved version of the source code format.

The CFG breaks the code into blocks of bytecode that are always performed as an uninterrupted group and establishes the connections that link the blocks together. The DFG maps the connections between where values are produced and where they are used. This includes connections within blocks of bytecodes and also connections between blocks of bytecodes. Because Java programs frequently use stack locations as implicit variables, tracking the propagation of values is a difficult process. When extracting the data flow information to generate the DFG for the IR, the prior art approach has been to perform a complete simulation of both the stack and iterations through all possible flow paths .

In contrast to traditional compilers where end users only interact with compiled programs, the run time nature of Java creates a significant incentive to optimize the speed and efficiency of Java optimizers and JIT compilers. Accordingly, it is an object of the present invention to provide a method for generating a DFG from the source

bytecodes to form an IR that optimizes the speed and efficiency of Java optimizers and JIT compilers.

These and many other objects and advantages of the present invention will become apparent to those of ordinary skill in the art from a consideration of the drawings and ensuing description of the invention.

SUMMARY OF THE INVENTION

The present invention is directed to the optimization of Java class files by improving the efficiency of generating a DFG for the IR.

According to a first aspect of the present invention, a forward scan of an uninterrupted bytecode block from a bytecode source file is performed to identify the start of each bytecode in the uninterrupted block. Next, a backwards scan is performed. For each bytecode in the block, a DFG node is created, links to all the values used by the bytecode in the block are established, and links to preceding or subsequent nodes in the file are created.

According to a second aspect of the invention, the bytecode blocks are linked in the file's order of execution and a stack state is generated for each block of code. For each join statement generated by the CFG, one path of the CFG is stepped through. Next, links are established in the DFG between locations where values are used and where the values are produced. Finally, for all blocks parallel to the blocks producing

values the links for bytecodes occupying the same stack state location are duplicated.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1A is a flow diagram of the optimization of a JAVA bytecode source file using a stand alone optimizer according to the present invention.

FIG. 1B is a flow diagram of the optimization of JAVA bytecode source file using a just in time compiler according to the present invention.

FIG. 2 is a flow diagram illustrating the generation of control flow graph from a bytecode source file suitable for use in the present invention.

FIG. 3 is a flow diagram of the generation of data flow graph according to the prior art.

FIG. 4 is a flow diagram of a first aspect of the generation of a data flow graph for the links in a block of bytecode according to the present invention.

FIG. 5A is a block diagram for the generation of a node in a data flow graph for the instruction IADD according to the first aspect of the present invention.

FIG. 5B is a block diagram for the generation of several nodes in a data flow graph for a plurality of instructions according to the first aspect of the present invention.

FIG. 6 is a flow diagram of a control flow graph suitable for illustrating a second aspect of the generation of a data flow graph for the links between blocks of bytecode according to the present invention.

FIG. 7A is a block diagram of the steps in the method according to the second aspect of the present invention.

FIG. 7B is a block diagram for the generation of nodes in a data flow graph according to the second aspect of the present invention.

DESCRIPTION OF THE PREFERRED EMBODIMENTS

Those of ordinary skill in the art will realize that the following description of the present invention is illustrative only and is not intended to be in any way limiting. Other embodiments of the invention will readily suggest themselves to such skilled persons from an examination of the within disclosure.

Referring first to FIGS. 1A and 1B, simplified flow diagrams 10 and 20 of the optimization of a Java bytecode source file are illustrated. In both flow diagrams 10 and 20, Java source code file 12 is compiled by a Java compiler 14 into Java bytecode classfiles 16. In flow diagram 10, the classfiles are operated on by a SAO 18, and in flow diagram 20, the classfiles are passed through a Java Runtime System 22 to a JIT compiler 24. The Java SAO 18 outputs Java classfiles 26 containing bytecodes that perform the

same operations as the input classfiles, only in an optimized manner. The JIT 24 outputs executable code 28 that can run directly on the native operating system 30. Those of ordinary skill in the art will recognize that the procedures described in FIG. 1 are merely illustrative and that there exist other structures within which classfiles may be optimized.

In processing an input bytecode source classfile, both an SAO and a JIT form an IR. As described above, an IR is a succinct and straightforward structuring of the control and data flow information of a program into a CFG and DFG. The CFG represents the information regarding the sequence and order of the execution of the bytecode instructions as blocks of code linked in the order of execution of the program. The blocks of code are sections of bytecode that are always performed as an uninterrupted group, and the links between the blocks of code are bytecode branch, conditional, or jump instructions.

FIG. 2 is a flow diagram that illustrates the generation of the CFG from a bytecode classfile 40. It should be appreciated that there are several known methods for generating a CFG. The specifics of any such method will not be disclosed herein to avoid overcomplicating the disclosure and thereby obscuring the present invention. The procedure begins with a classfile 40 containing an ordered sequence of bytecode instructions numbered from 1 through N. The numbers 1 through N are analogous to the numbering of lines in a computer program as is known in the art. Next, an optimizer 42 processes the bytecode classfile 40 to extract the control flow information from the bytecodes.

The control flow information is illustrated by the CFG 44. For example, the CFG 44 may indicate that for classfile 40 that bytecode instructions 1 through 3 are first executed, instructions 100 through 103 are next executed, either instructions 10 through 14 or 15 through 19 are next executed, and finally instructions 4 through 8 are executed before the remainder of the bytecode instructions execute. It will be appreciated that the CFG 44 is an example only, and that any number of CFGs may be generated from the classfile 40 by the optimizer 42.

As described above, once the CFG has been generated, the DFG will be generated. According to the present invention, two aspects of the DFG are generated from the CFG. In one aspect, the propagation of values within blocks of bytecode is determined. As is known in a typical JAVA program, data values are produced in one location and consumed in another. The first aspect of the present invention relates to establishing the connections between the production and consumption points that occur within uninterrupted blocks of code. The present invention allows implicit variables to be easily recognized and for their values to be immediately linked in the DFG to the bytecode(s) using them.

Unlike the prior art, the present invention avoids the approach taken in the prior art of constructing a stack to determine the values used by bytecodes. For example, in FIG. 3, a block diagram of a known method for generating links within a code block of a CFG is illustrated. The diagram demonstrates the principle stages employed to determine

the value of implicit stack location variables when generating a DFG. During stage 60, the operations of an input sequence of bytecodes 62 are simulated to create a version of the stack 64 that they represent. The stack shown 64 presents only a portion 66 of the input sequence of bytecodes 62. During stage 68, the stack 64 is executed, and it is determined that the POP instruction 70 is associated with the value "2" due to the preceding BIPUSH "2" 72 instruction. This information is stored in a DFG node represented by reference numerals 74 and 76.

According to the preferred embodiment of the present invention, a node tree is employed. When values are produced by the bytecode classfile, a new node is created in the tree. This node is then connected to the rest of the tree by linking it to the value's production and consumption points, as they become known. In this manner, the propagation of data values through a program can be readily traced. Those of ordinary skill in the art will recognize that a variety of data structures could be used.

Turning now to FIG. 4, a flow diagram of a method for generating links within a bytecode block according to the present invention is illustrated. In a first step 100, a forward scan of a bytecode source file 102 is performed in order to determine the starts 106, 108, and 110 of the bytecodes 112, 114, and 116, respectively, because as is known, bytecodes can be of different lengths. Next, at step 118 a backward scan of the bytecode source file 102 is performed. During the backward scan step 118, the nodes of the DFG are generated as they occur. For example, if bytecode 116 is the instruction POP, as POP is read, a node 120 is created in the DFG including a POP instruction 122.

If next, the instruction BIPUSH "2" is read at bytecode 108, the value "2" at 124 is linked to the POP instruction 122 that consumes it.

The present invention provides a significant increase in the efficiency in the generation of the DFG because the entire process of constructing a copy of the stack has been eliminated. The invention recognizes that because the stream of bytecodes are the instructions for a stack machine, the operands or values consumed by a bytecode instruction precede the instruction. When the bytecode file is read in reverse order, the step of creating a node in the DFG can be immediately performed after the bytecode is read.

Further examples, not intended to be exhaustive, are illustrated in FIGS. 5A and 5B of the nodes in a DFG created from bytecode instructions according to the present invention. FIGS. 5A and 5B respectively are examples of a single instruction with two links, and a series of instructions with multiple links.

In FIG. 5A, a bytecode source block 150 has been first differentiated by a forward scan into bytecodes 152, 154 and 158. When the backward scan step is performed, an integer addition (IADD) instruction is read from bytecode 158, and a node 160 is created in the DFG. From the definition of the instruction IADD it is known that node 160 will consume two values. As the backward scan continues, these two values are read from bytecodes 154 and 152, created as nodes 162 and 164 in the node tree of the DFG, and then linked to node 160.

In FIG. 5B, a bytecode source block 166 has been first differentiated by a forward scan into bytecodes 168, 170 and 172. The sequence of bytecodes 166 implements the statement (X + X). In the backwards scan step the first bytecode 172 encountered is an IADD instruction. As explained with respect to the example of FIG. 5A, the IADD instruction generates a node 174 in the DFG with links to two values. Since the next bytecode 170 is a duplicate (DUP) instruction both of the links from node 174 are made to the node 176 for the DUP instruction. Next, bytecode 168 having an integer load X (ILOADX) instruction is read, and a node 178 with a link to node 176 is created. In this sequence in the node tree, the value in node 178, will be duplicated in node 176, and both values will be consumed in node 174. As will be appreciated by those of ordinary skill in the art from the above recited examples, when performing the backwards scan step, the particular DFG created will depend upon the sequence of bytecodes in the blocks generated by the CFG.

According to a second aspect of the present invention, the propagation of values between blocks of bytecode in the CFG is determined. FIG. 6 illustrates a CFG 180 having several blocks of linked code 200, 202, 204, and 206. In the CFG 180, the blocks 202 and 206, that precede a join 208 are considered parallel. It should be appreciated that a join may connect a plurality of parallel blocks. When the CFG is generated, stack states, 210, 212, 214, and 216 are created for each block of bytecode 200, 202, 204, and 206, respectively. The stack states includes all the values still present on the stack after the bytecodes in the block associated with that stack have been

整理番号 162880

SUNP-2412

executed. It should be appreciated that values on the stack are produced by instructions in the bytecode blocks. This is shown by the mapping of the instructions 218 and 220 to the stack states 222 and 224, respectively. For purposes of illustration to be described below, instruction 218 is PUSH "1", instruction 220 is PUSH "0", and instruction 226 is POP.

To determine when a value produced in an early block of bytecode is used or consumed by a subsequent block of bytecode to generate a link in the DFG between blocks of bytecode, an execution of the blocks is made while noting the consumption of a prior stack state value and identifying its source. This information is represented in the DFG as a link between the nodes of the consumption and production of the bytecodes. To establish links for all parallel blocks, the method in the prior art relied upon iterating through all possible control flow paths in the CFG. For example, to generate the DFG for links between blocks in the CFG 180, a first pass would be made through the path including blocks 200, 202 and 204, and then a second pass would be made through the path including blocks 200, 206 and 204.

According to the present invention, not all control flow paths must be traversed to form the DFG for links between parallel blocks of bytecode. Rather, the links in the DFG between blocks 200, 202, 204 and 206 can be generated simply by stepping through blocks 200, 202, 204 and then applying the information from the pass through blocks 200, 202, 204 to establish the links to block 206. The present invention exploits the fact that Java standards' require and the verification described above ensures that in

properly formatted code, the stack states for any of the blocks preceding a join are the same.

For example, in CFG 180, both of the stack states 212 and 216 that precede the join 208 must be the same in both size and relative type. During execution, the consumption of any value in block 202 will be paralleled by a consumption of a value in block 206. As a result, for example, if the value corresponding to the stack state location 222 is used by the bytecode instruction in 226 of code block 204, then the bytecode instruction 226 also uses the value corresponding to the stack state location 224.

The following lines of pseudocode illustrate the point.

```
IF (some condition occurs)
        X = 1;
ELSE
        X = 0;
•
•
•
A = X + 2;
```

The assignment of a value to "A" in the expression "A = X + 2" uses the value from either expression "X = 1" or X = 0" that has been assigned to the implicit stack variable "X" as determined by the "IF (some condition occurs)" statement.

In FIG. 7A, a block diagram outlining the steps employed to generate a DFG according to the second aspect of the present invention is illustrated.

At step 250, a first branch of a CFG graph is scanned. Applied to CFG 180 in FIG. 6, a first pass through blocks 200, 202 and 204 is performed.

Next at step 252, links between the place a value is created and the place a value is consumed are made. As illustrated in FIG. 7B, a link 254 is made between the PUSH "1" instruction at location 218 in the CFG 180 that the POP instruction consumes at location 226 in the CFG 180.

Next at step 256, a link 258 is made between the PUSH 0 instruction at location 220 that the POP instruction consumes at location 226, because it is known that the data for this instruction occupies the same respective stack state location as the PUSH 1 instruction at location 218 in the CFG 180. As such, without traversing all paths in the CFG as relied upon by the prior art, according to the second aspect of the present invention, the links in a DFG between blocks of bytecodes can be more efficiently made. As joins are quite common throughout bytecode source code, the present invention represents a significant increase in efficiency. Where multiple joins exist, the increase in efficiency is even greater.

While illustrative embodiments and applications of this invention have been shown and described, it would be apparent to those skilled in the art that many more

modifications than have been mentioned above are possible without departing from the inventive concepts set forth herein. The invention, therefore, is not to be limited except in the spirit of the appended claims.

整理番号 162880

SUNP-2412

CLAIMS

What is claimed is:

1. A method for linking bytecodes of an uninterrupted block of bytecodes in the formation of a data flow graph comprising the steps of:

scanning said uninterrupted block of bytecodes in a forward manner to identify the start of each of said bytecodes;

scanning in a backward manner bytecodewise each of said bytecodes in said uninterrupted block of bytecodes; and

generating a link in said data flow graph that links each of said bytecodes to all other of said bytecodes used by said each of said bytecodes.

2. A method for linking bytecodes between uninterrupted blocks of bytecodes in the formation of a data flow graph, said uninterrupted blocks of bytecodes having links according to an order of execution of said uninterrupted blocks and wherein a stack state has been generated for each of said uninterrupted blocks of bytecodes, comprising the steps of:

stepping through a first path of a plurality of paths of said order of execution that terminates in a join to generate a link in said data flow graph between each bytecode producing a value in one of said uninterrupted blocks and each bytecode consuming said value in another of said uninterrupted blocks in said first path; and

duplicating each link in said first path with a link for each bytecode in all of said

plurality of paths other than said first path for each bytecode producing a value having a similar stack location to each bytecode producing a value in one of said uninterrupted blocks in said first path.

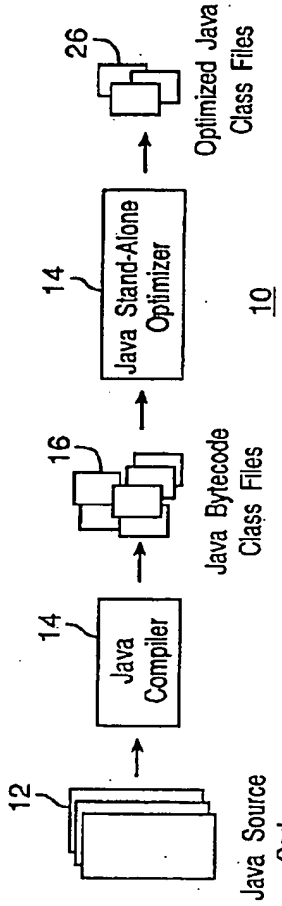

12
Java Source Code
14
Java Compiler
16
Java Bytecode Class Files
14
Java Stand-Alone Optimizer
26
Optimized Java Class Files
10

Fig.1A

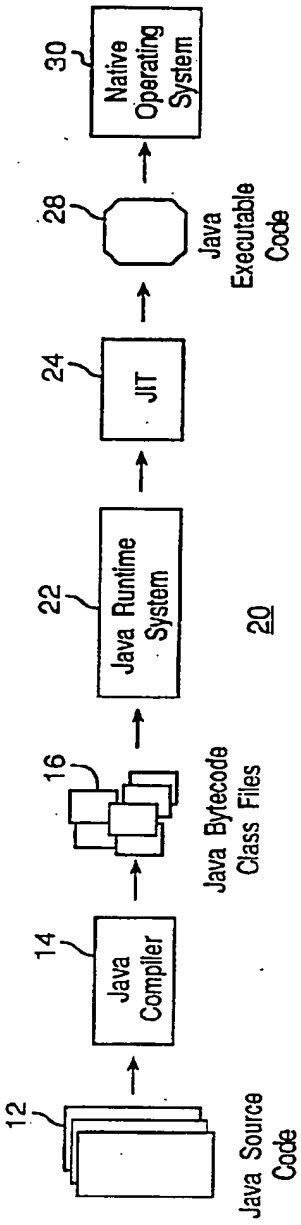

12
Java Source Code
14
Java Compiler
16
Java Bytecode Class Files
22
Java Runtime System
24
JIT
28
Java Executable Code
30
Native Operating System
20

Fig.1B

整理番号　１６２８８０

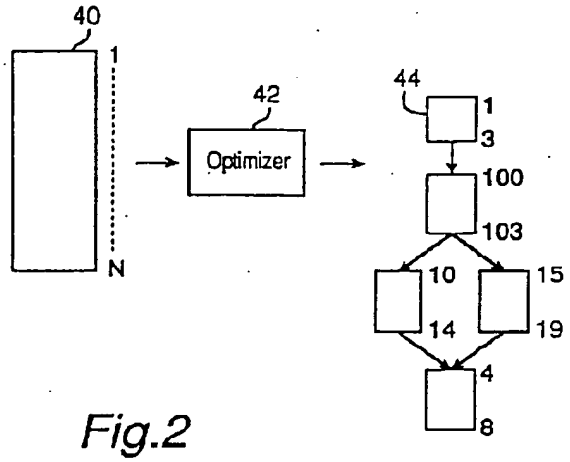


Fig.2

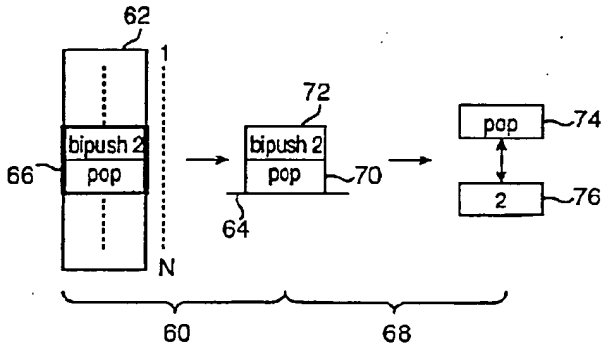


Fig.3

整理番号 １６２８８０

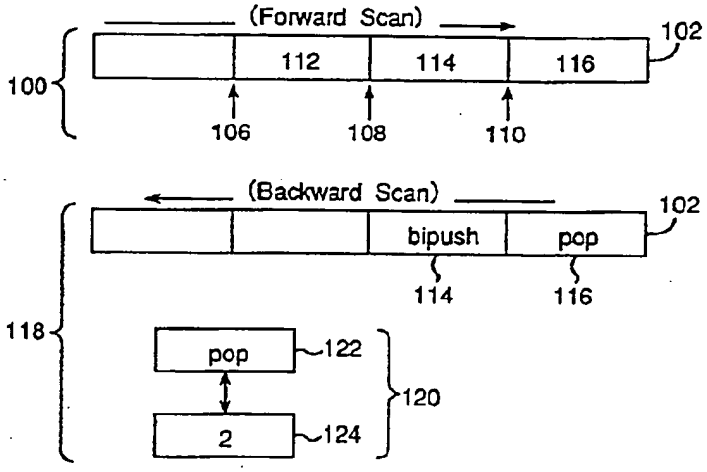


Fig.4

整理番号 １６２８８０

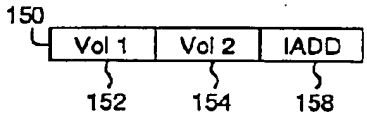
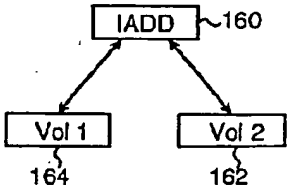


Fig.5A



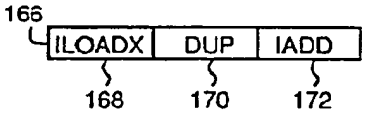
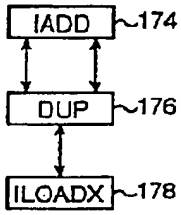


Fig.5B



整理番号　１６２８８０

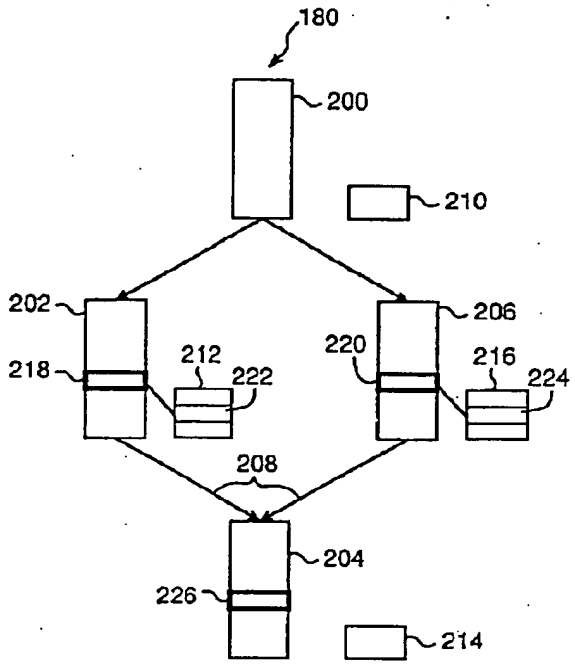


Fig.6

整理番号 162880

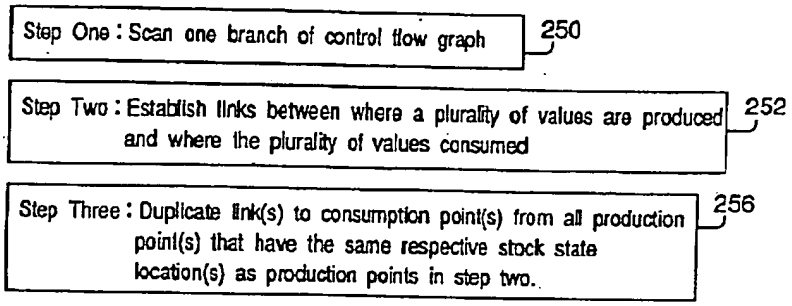


Fig.7A

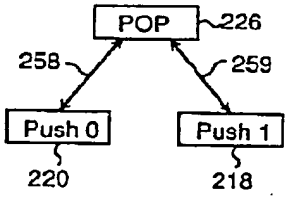


Fig.7B

ABSTRACT OF THE DISCLOSURE

According to a first aspect of the present invention, a method for linking bytecodes of an uninterrupted block of bytecodes in the formation of a data flow graph comprises the steps of scanning the uninterrupted block of bytecodes in a forward manner to identify the start of each of the bytecodes, scanning in a backward manner bytecodewise each of the bytecodes in the uninterrupted block of bytecodes, and generating a link in the data flow graph that links each of the bytecodes to all other of the bytecodes used by the each of the bytecodes.

According to a second aspect of the present invention, a method for linking bytecodes between uninterrupted blocks of bytecodes in the formation of a data flow graph, the uninterrupted blocks of bytecodes having links according to an order of execution of the uninterrupted blocks and wherein a stack state has been generated for each of the uninterrupted blocks of bytecodes, comprises the steps of stepping through a first path of a plurality of paths of the order of execution that terminates in a join to generate a link in the data flow graph between each bytecode producing a value in one of the uninterrupted blocks and each bytecode consuming the value in another of the uninterrupted blocks in the first path, and duplicating each link in the first path with a link for each bytecode in all of the plurality of paths other than the first path for each bytecode producing a value having a similar stack location to each bytecode producing a value in one of the uninterrupted blocks in the first path.